User's Manual for Version 1.1 of the NIST DME Interface Test Suite for Facilitating Implementations of Version 1.1 of the I++ DME Interface Specification

John Horst, Thomas Kramer, Joseph Falco, Keith Stouffer,
Hui-Min Huang, Cathy Liu, Frederick Proctor, and Albert Wavering
The National Institute of Standards and Technology (NIST)
U. S. Department of Commerce
Gaithersburg, Maryland, USA
Telephone: 301.975.{3430, 3518, 3455, 3877, 3427, 3434, 3425, 3461}
Email: {john.horst, thomas.kramer, joseph.falco, keith.stouffer, hui-min.huang,
cathy.liu, frederick.proctor, albert.wavering}@nist.gov

May 9, 2003

"In software development, testing should occur:

1. At the beginning
2. In the middle
3. On those who think testing should occur only at the end"

–a poster advertisement for Rational Software Corporation

Contents

List of Figures

List of Tables

1 Version information

This user's manual describes version 1.1 of the NIST Dimensional Measuring Equipment (DME) interface test suite intended to support implementations of version 1.1 of the I++ DME specification. This and other versions of the I++ DME specification are located at the following web site.

`http://www.isd.mel.nist.gov/projects/metrology_interoperability/specs/index.html`

2 Quick startup instructions: Getting started with the client-side and server-side utilities and test cases

The following instructions are for those developers who want to immediately start using the NIST test utilities and test cases. This includes those who are writing their own code from scratch and do not intend to use any of the code provided by NIST in their implementations.

2.1 Downloading and unpacking the test suite

A zip file, containing all the source code, executables, and additional files pertaining to the test suite, can be downloaded from the following link:

`http://www.isd.mel.nist.gov/projects/metrology_interoperability/resources.html`

The zip file, NISTutilitiesVer1.1ForIppVer1.1.zip, contains a directory called NISTI++DMEtestSuite1.1. The high level directory structure of the file when it is unzipped is shown in Figure 1.

2.2 Running the server-side and client-side utilities

The server-side and client-side utilities are located here:

```
NISTI++DMEtestSuite1.1\testSuiteUtilities\client.exe
NISTI++DMEtestSuite1.1\testSuiteUtilities\server.exe
```

These executable files, client.exe and server.exe, are for testing server-side and client-side implementations of the I++ specification, respectively. For information on how to run client.exe, see Section 7.1. For information on how to run server.exe, see Section 8.1.

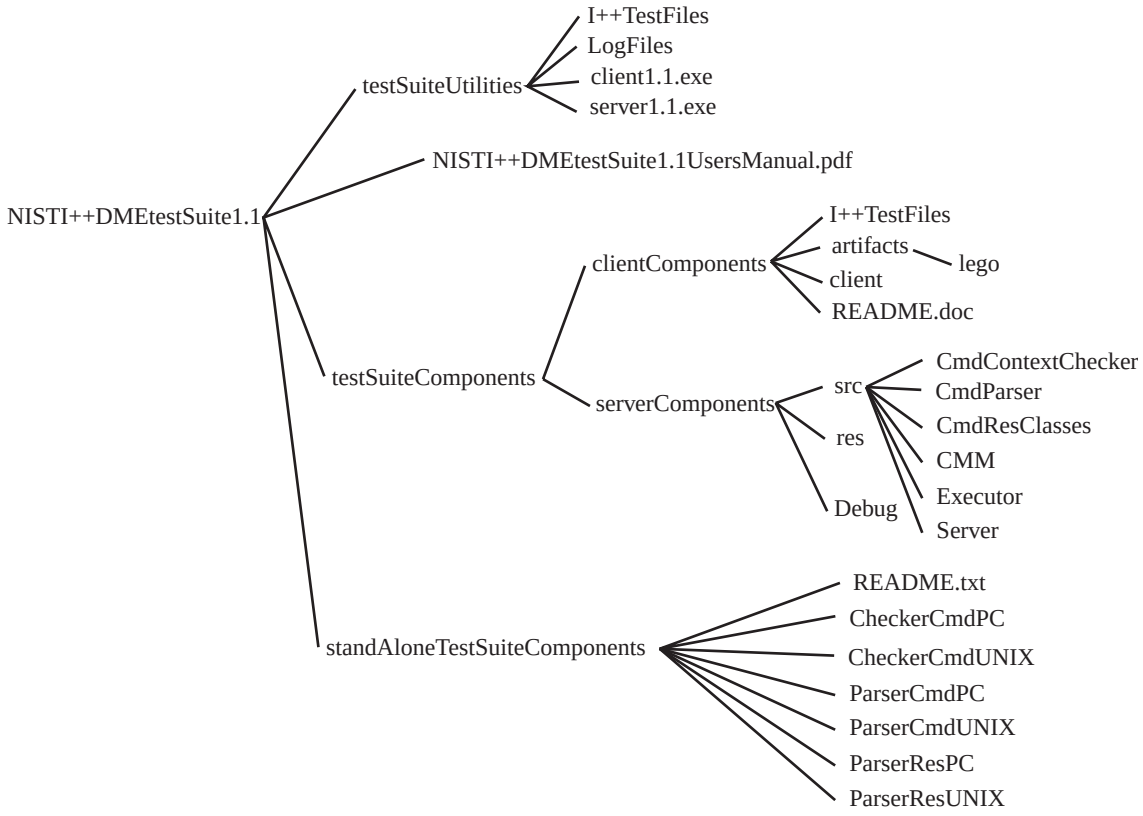

Figure 1: Directory structure for the NIST test suite version 1.1

2.3 Using the test cases

The test cases currently consist of two items, 1) test files of I++ command strings and 2) a test artifact. The test files and files describing the artifact are included in the test suite and are located in the following directories:

```
NISTI++DMEtestSuite1.1\testSuiteComponents\clientComponents\I++test_files
NISTI++DMEtestSuite1.1\testSuiteComponents\clientComponents\artifacts\lego
```

How to use the test files is described in Section 7.4 and how to purchase, assemble, and provide Computer-Aided Design (CAD) support for the Lego[1] artifact is described in Section 7.5.

The test cases described above are only for testing server-side implementations. Currently, only a single test case exists for testing client-side implementations...it provides correct responses only. We plan to add more test cases by providing incorrect responses in order to more fully test client-side implementations.

3 Platform requirements

All code in this test suite is intended to be developed and run on a PC running Microsoft Visual C++ version 6.0 Professional. The server-side and client-side utilities, as well as the stand-alone component executables (see Section 6.2, created in Visual C++ should run under all Windows-based operating systems (at least on Windows 95 and all subsequent versions). Certain software components in the test suite, namely, the graphical user interface (GUI) front end and the Transmission Control Protocol/Internet Protocol (TCP/IP) socket read/write component, are heavily dependent on the Visual C++ platform constraint. Others such as the parser and the context checker are not. The socket facility is Winsock version 1.1.

[1] Certain commercial companies and their equipment, instruments, or materials are identified in this paper in order to specify the experimental procedure adequately. Such identification is not intended to imply any judgment by the National Institute of Standards and Technology concerning the companies or their products, nor is it intended to imply that the materials or equipment identified are necessarily the best available for the purpose.

4 How to get the most out of the entire test suite

This test suite is currently designed to *facilitate* the creation of I++ DME Specification-compliant implementations and not to provide compliance tests. Compliance (and interoperability) testing is planned for a second phase of test suite development. The test suite currently facilitates both client and server side implementations and we suggest that it be used in the following three ways:

1. provide utilities that can be run against an implementation to validate correct performance and help speed up debug efforts
2. make available all test suite components which can be integrated into anyone's implementation, which will
   (a) shorten implementation development times
   (b) tend to standardize and simplify implementations, further ensuring successful system interoperability in the end
3. provide a benchmark interpretation of the specification that will be used to clarify and disambiguate the specification itself

We now describe these three ways of using the test suite in more detail.

4.1 Using the executables to facilitate implementation development

This type of use is described in Section 2.

4.2 Using software modules to facilitate implementation development

The intent of this test suite is not only to provide utilities that will ultimately test implementations for compliance, but also to provide code modules, such as command and response classes, command and response string parsers that can be used by implementors of the I++ DME specification. We believe that the use of this code can greatly facilitate implementation development (on both client and server sides of the DME interface). Even more so, we anticipate that the use of common code like that provided in the NIST DME test suite, will greatly speed up the testing of implementations, allow the specification to grow into a quality, sufficiently functional, and unambiguous specification, and finally will allow the specification to move into a standards phase much more quickly.

For example, all developers of server-side implementations need a command-string parsing engine of some sort in their code. However, if all implementors agree to use the same command-string parsing engine (parsing being a highly modularizable function), *all* can make improvements to it and when testing is complete, the parsing code will be of high quality. It still would not be required by the specification, but simply helpful for developing efficient implementations. The same arguably goes for several other parts of the NIST test suite, including modules for socket read/write, command context checking, command classes, response classes, log file generation, and response context checking. These and other modules should not be required as standard by the specification, but if employed as common code in *all* implementations, a higher quality specification and ultimately a higher level of interoperability will be achieved.

4.3 Using the test suite to improve the specification

An unambiguous, tightly-defined, and fully functional specification is essential to ultimately ensure interoperability between various implementations. Undoubtedly, as we develop implementations and use the test suite, weaknesses within the specification will be revealed. What better time than during the development and testing phase to correct these weaknesses...before the specification is released as an international standard?

5 Using the test suite for conformance and interoperability testing

It is NIST's *ultimate* intent to have this test suite be expanded to support actual conformance tests and interoperability tests, *i.e.*, the components of this expanded test suite will be used to test the conformance of any single implementation of the specification and finally to support interoperability tests on pairs of implementations. Which is to say that the test suite provided does not currently constitute a formal test suite, since they are not accompanied by any test metric, test results analysis tools, or test procedures. More importantly, we have not defined enough test cases, providing sufficient coverage, to qualify the test suite as suitable for conformance.

Though we are not currently providing conformance tests, we do include a preliminary set of test cases (see Section 7.4) for facilitating both server-side and client-side implementation development. The client-side test cases consist of a test artifact and test inspection programs. The test inspection programs consist of text files of I++ DME compliant command strings and a specification

for the format of that text file. One of the server-side components is a single test case for responses. This test case always produces correct responses, but we have placeholders for adding more test cases which can create incorrect responses of various types. Our client-side utility also produces a log file to facilitate debugging and ultimately to use for testing compliance.

In an earlier version of the client-side utility for a prior DME specification development effort, the client-side utility was used to run a DME command file on a coordinate measuring machine (CMM) located at a facility in England (LK Metrology, Ltd.) remotely from NIST in Gaithersburg, Maryland, USA. For that demonstration, a user-controlled pan/tilt/zoom camera was integrated into the environment with a web-based video server to allow the inspection program to be viewed from NIST. We envision this capability to be particularly useful for future interoperability testing, but also for conformance testing.

6 Introducing the components of the test suite

6.1 The two sets of components within the test suite: client-side components and server-side components

The test suite currently consists of two separate sets of components.

- Client-side components for facilitating the development of server-side (*i.e.*, the equipment side of the interface) implementations. Table 1 lists all the client-side components and gives a brief description of the operation of each component.
- Server-side components for facilitating client-side (*i.e.*, the application software side of the interface) implementations. Table 2 lists all the components of the server-side components and gives a brief description of the operation of each component.

The operation of these two sets of components, both as a test suite and as actual implementations, is illustrated in Figure 2. In this figure we see the components of the test suite that can be used in actual implementation code and those that normally will not. We want to encourage implementors to particularly consider the use of the command and response classes, which encapsulate so much interpretive knowledge about the specification that we believe it will be helpful in assuring interoperability between those implementations that use the same set of command and response classes.

6.2 Stand-alone components versus stripped-down components

All the components listed in Tables 1 and 2 are located in NISTI++DMEtestSuite1.1\testSuiteComponents\. This directory is intended to be used by those developers that 1) want to bring up the entire client-side or server-side project (or workspace) within the Visual C++ environment and immediately begin editing, compiling, and linking or 2) developers that want to include some or all of these components in their implementations. Some of the subdirectories do not contain source code, for example, the artifacts subdirectory. The source code files in NISTI++DMEtestSuite1.1\testSuiteComponents\ are compiled and linked into either the client-side or server-side utility. Therefore, they are stripped-down versions of the components, in the sense that they cannot be compiled and executed alone.

Several of the components have stand-alone versions with main() functions. These components are located in NISTI++DME testSuite1.1\standAloneTestSuiteComponents\. These will be helpful 1) to instruct an implementor how the component needs to be called within the implementor's code and 2) in certain cases, the stand alone code can be used to test the validity of command and response strings files.

6.3 Who to contact for a particular test suite component

In order to best use the test suite components, it's good to know who to contact when questions arise. Table 3 identifies the individuals responsible for each component's development and maintenance (all of these individuals are NIST staff members and co-authors of this user's manual).

We will now describe how to fully utilize these components for developing quality implementations in an expeditious manner.

7 Using the client-side components for facilitating server-side implementations

Here's a summary of the operation of the client-side utility. The I++ DME commands are sent over a TCP/IP socket to an I++ DME compliant server. The server receives the commands, executes them, and returns the appropriate response back to the client-side utility via the socket. The client-side creates a time-stamped logfile of I++ DME commands sent and responses received over the socket.

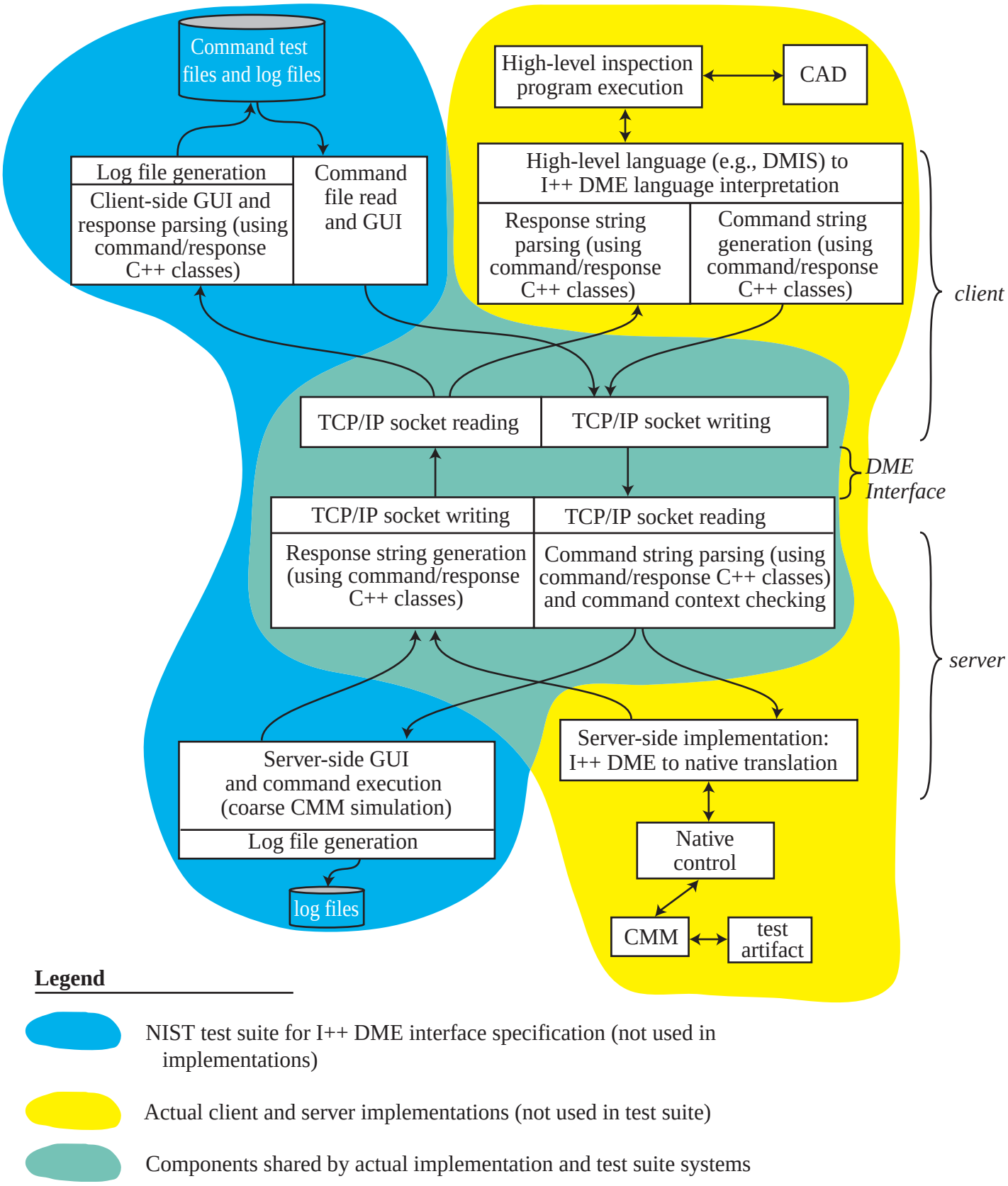

Figure 2: I++ DME Interface Specification Testing and Implementation Systems

Table 1: A list of all the client-side components along with a brief description of what each does and where it is located

Client-side component	Description	Directory location
GUI front end	Provides a front end to the implementation developer that is easy to comprehend and use and also contains the "glue" code that integrates all client side components	NISTI++DMEtestSuite1.1\ testSuiteComponents\ clientComponents\client
TCP/IP socket read and write	Writes command strings and reads response strings from the specified TCP/IP channel	This component is currently integrated into the GUI front end code (client). Modularization is planned for a later version. However, it can currently be easily extracted from client.
Response parser	As a complement to the command parser, it receives response strings as input and outputs either an error message or an instance of the appropriate response class stuffed with the values extracted from the response string	a response parser is not required for operation of the current client-side utility, since no context checking is done, but a stand-alone response parser can be found in NISTI++DMEtestSuite1.1\ standAloneTestSuiteComponents\ ParserResPC
Client-side log file generator	The log file records time-stamped versions of all commands sent and responses received	Currently integrated into the GUI front end. Modularization is planned for a later version. Furthermore, it can be easily extracted from client.
Test command files	These test files consist of strings of I++ DME compliant commands. Currently consist of the following subsets of test files: 1) test files in which each contains one and only one command, 2) test files with multiple commands, 3) test files with syntax errors, 3) test files with semantic errors, and 4) test files for real execution on the Lego artifact	NISTI++DMEtestSuite1.1\ testSuiteComponents\ clientComponents\I++test_files
Test artifact	Consists of an assembly constructed from Lego parts, where all these parts are contained in Lego sets currently available on www.lego.com. The test case contains artifact assembly instructions, a CAD representation, and a shopping list for the required parts. Provides low cost, high malleability, and high worldwide availability at the cost of low measurement accuracy	NISTI++DMEtestSuite1.1\ testSuiteComponents\ clientComponents\artifacts\lego

The client-side components provide developers with a GUI allowing the developer easy control over a number of parameters, including the choice of the TCP/IP socket channel number, the choice of inspection programs (test files) to run, and the choice of whether to type in commands by hand, run test files *in toto*, or run test files line by line. The client-side GUI also generates a log file that is helpful for debugging. The log file contains time-stamped versions of all commands sent and responses received for each test file run. The client-side utility checks that the response is consistent with the command sent for matching tag numbers between command and response. The client-side GUI contains a response parser, however we have developed new response parser that is more consistent with the command parser in the server-side utility (described in Section 8). This new response parser will be integrated into the client-side utility in a later version of the test suite.

In summary, the client-side components consist of the following components. Table 1 gives a description of each of the components listed here:

- GUI front end
- TCP/IP socket read and write
- Response parser
- Log file generator
- Test cases consisting of

Table 2: A list of all the server-side components along with a brief description of what each does and where it is located.

Server-side component	Description	Directory location
TCP/IP socket read and write	Reads command strings and writes response strings from the specified TCP/IP channel	Currently integrated into the GUI front end. Modularization is planned for a later version. Furthermore, it can be easily extracted from client.
Command parser	Receives command strings as input and outputs either an error message or an instance of the appropriate command class stuffed with the values extracted from the response string	NISTI++DMEtestSuite1.1\ testSuiteComponents\ serverComponents\CmdParser
Command context checker	Looks at the current command in light of the commands previously executed. If the context is illegal, an error is sent. If the context is legal, the command is sent to the executor for execution	NISTI++DMEtestSuite1.1\ testSuiteComponents\ serverComponents\CmdContextChecker
Command executor (CMM simulator)	Executes commands and generates responses according to the test cases selected in the server-side GUI	NISTI++DMEtestSuite1.1\ testSuiteComponents\ serverComponents\Executor
Server-side GUI front end	Provides a front end to the implementation developer that is easy to comprehend and use	NISTI++DMEtestSuite1.1\ testSuiteComponents\ serverComponents\Server
Command and response C++ classes	Defines all the data structures and methods necessary for building instances of command and response, including the ability to generated lists (queues) of commands and responses	NISTI++DMEtestSuite1.1\ testSuiteComponents\ serverComponents\CmdResClasses
Server-side utility	Integrates the command parser, the context checker, the command and response class instances, the server-side GUI front end, the command executor, and the socket read/write facility into a single executable utility	NISTI++DMEtestSuite1.1\ testSuiteComponents\ serverComponents\Server
World model	Contains data and methods for storing and maintaining information about the state of the CMM and its environment	NISTI++DMEtestSuite1.1\ testSuiteComponents\ serverComponents\CMM

- Test files of I++ command strings (some with errors and some without)
- A Lego artifact consisting of assembly instructions, CAD representation, and parts shopping list

7.1 Operating the GUI front end

Figure 3 illustrates the client-side GUI. To execute the client-side utility, the user first selects which type of file to run (I++ DME or DMIS (Dimensional Measurement Interface Standard)) and then enters the name of the file to run. When a DMIS file is selected, the file is run though an interpreter that converts the DMIS command to the appropriate I++ DME command(s). Only a select group of DMIS commands are supported at this time. The user then enters the part transformation information representing the translation and rotation required to go from machine to part coordinates, since test file position values are in the part coordinate system. For the Lego artifact, the part origin is at the outside corner of the "tower" section (the section containing the highest point on the artifact), at the base of the entire artifact, with the x-axis along the long side of the tower, the y-axis along the short side, and the z-axis pointing up. The user then selects the name of the logfile where the time-stamped data will be recorded.

The user specifies the Hostname of the I++ DME compliant server to connect to. The Hostname can be entered as either fully qualified hostname or IP address. The user then specifies the port number to create the socket with. The default port number, 1294, is the one given in the I++ DME specification. When the user clicks on the "Connect to CMM Controller" button, a non-blocking TCP/IP socket is created between the client-side utility and the CMM server on the specified port. Once the client-side utility is connected to the server, the user can either enter a command manually, single step through the program file that was selected, or run the entire file. A status window displays the current status of the program, including the command just sent or response received as well as the existence of any error conditions.

Table 3: A list of the test suite components and the individuals responsible for them

Test Suite Components	**Individuals Responsible for Component Development and Maintenance**
Client-side GUI front end	Keith Stouffer
Client-side utility	Keith Stouffer
TCP/IP socket read and write	Keith Stouffer
Command parser	Tom Kramer
Response parser	Tom Kramer
Client-side log file generator	Keith Stouffer, Hui-Min Huang, and Cathy Liu
Test command files	Tom Kramer and Cathy Liu
Test artifact	John Horst
Command context checker	Tom Kramer
Command executor (CMM simulator)	Cathy Liu, Joe Falco, and Hui-Min Huang
Server-side GUI front end	Keith Stouffer
Command and response C++ classes	Joe Falco, Tom Kramer, and John Horst
Server-side utility	Joe Falco, Tom Kramer, Hui-Min Huang, and Cathy Liu

7.2 Using the TCP/IP socket read and write code

The code for reading from and writing to the TCP/IP socket is common for both client and server side components. Since we are restricting implementation platforms to be a PC with a Windows operating system, the socket read/write code component can be easily integrated into implementation code, saving the implementor much time and effort. So, the socket read/write component in the test suite is one of those components that is used in the executing utilities (both server and client side) and can be used in implementation code, as is illustrated in Figure 2.

7.3 Using the log file generation code

The log file generation code is modularizable, but not currently defined as a separate module. We plan to add this in our next version of the test suite.

However, if any implementors wish to extract the log file generation code themselves, they are able to. Simply access the file, ServerDlg.cpp located in

`\NISTI++DMEtestSuite1.1\testSuiteComponents\serverComponents\src\Server`

7.4 Using the command string test files

Types of test files: The test files consist of lines of command strings. Our current set of test files fall into four separate categories. The four categories with the corresponding files that belong to each category are listed in Table 4.

We now describe the format and content of these files. This allow you to develop and be able to share your own test files with all other developers.

7.4.1 Format of command strings

This file uses the syntax requirements previously proposed for I++ DME Interface command strings by NIST. The requirements on tags in the proposal have been modified to reflect the decision made at the July 3, 2002 meeting in Frankfurt, Germany that the only requirement on tags beyond their basic format be that a tag be unique among currently active tags.

This document covers the syntax of individual command strings. For a command string to be legal in a session, it is necessary but not sufficient that its syntax conform to the requirements given here. Constraints on the situations in which a command string may be given exist, but are not included here.

Any string purported to be a command string but failing to conform to the requirements given here should cause an error of some type to be issued by the server.

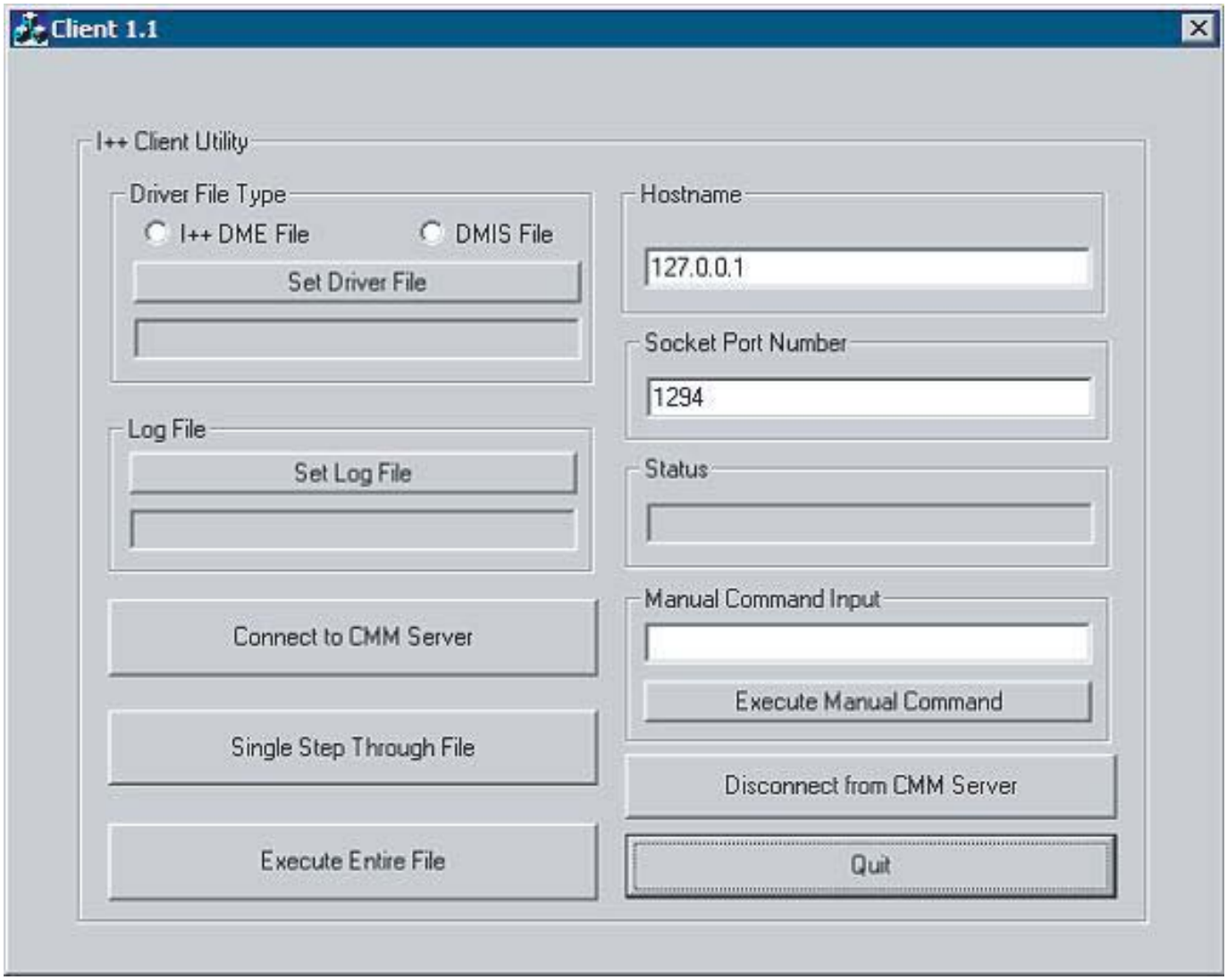

Figure 3: The client-side GUI front end

1. What is an I++ Command String?

   An I++ command string is a string of characters intended to be put into a character string to be transmitted by a client through a communications system to an I++ DME Interface server. The command string represents an I++ DME Interface command. See section 7.4.2 for details on command files and character strings.

2. Use of ASCII

   All references to ASCII characters in this command string spec are given using decimal (not octal or hex) numbers. An ASCII number enclosed in angle brackets (e.g., <13>) is used to represent ASCII characters.

3. Character Set

   All bytes in a command string are to be interpreted as 8-bit ASCII characters. The 8th bit must always be zero.

   Only characters in the range from <32> (space) to <126> ( ) may be used, except that the character pair carriage return <13> and line feed <10> is used as a command string terminator.

   The characters <13> and <10> may not occur anywhere else in a command string other than at the end, and in the order <13><10>.

Table 4: A list of all the command string test files by type of test

Type of test file	**Test file names**
Test files that each contain one or more instances of one type of command (plus instances of any other commands required to establish a context or to make the file usable without error in the client utility)	AbortE.prg, AlignTool.prg, ChangeTool.prg, ClearAllErrors.prg, DisableUser.prg, EnableUser.prg, EndSession.prg, EnumAllProp.prg, EnumProp.prg, FindTool.prg, Get.prg, GetCoordSystem.prg, GetCsyTransformation.prg, GetErrorInfo.prg, GetErrStatusE.prg, GetMachineClass.prg, GetProp.prg, GetPropE.prg, GetXtdErrStatus.prg, GoTo.prg, Home.prg, IsHomed.prg, IsUserEnabled.prg, OnMoveReportE.prg, OnPtMeasReport.prg, PtMeas.prg, ReQualify.prg, SetCoordSystem.prg, SetCsyTransformation.prg, SetProp.prg, SetTool.prg, StartSession.prg, StopAllDaemons.prg, StopDaemon.prg
test files that contain some syntactical errors in command strings	numbers_error.prg, parser_errors.prg
test files that contain some logical or semantic errors in the command strings	checker_errors.prg
test files that contain multiple correct command strings	numbers_ok.prg, all_ok.prg
test files that contain commands for use on the test artifact	lego.prg

Upper case letters and lower case letters are regarded as different letters. In other words, command strings are case sensitive.

4. Command String Length

   The maximum number of characters in a command string, including the <CR><LF>, must not exceed 65536.

5. Numbers

   Numbers are defined in the following hierarchy.

   number
   - integer
     - unsigned_four_digit_integer
   - decimal_point_number
     - no_exponent_number
     - exponent_number

   A digit is one of the characters 0 to 9 (<48> to <57>).

   An unsigned_four_digit_integer consists of exactly four digits. Example: 0287

   An integer consists of an optional plus <43> or minus <45> sign followed by one to sixteen digits. Example: +287741

   Note that every unsigned_four_digit_integer is also an integer.

   A no_exponent_number consists of an optional plus or minus sign followed by either: a. a decimal point <46> followed by one to sixteen digits, or b. one or more digits followed by a decimal point followed by zero or more digits so that the total number of digits before and after the decimal point is not more than sixteen. Examples: (a) -.3090 (b) 5.31

   An exponent_number consists of an optional plus or minus sign followed by one non-zero digit followed by a decimal point followed by zero to fifteen more digits followed by an E <69> or an e <101> followed by an optional plus or minus sign followed by exactly two digits. Example: -2.8843e02

   A decimal_point_number is either an exponent_number or a no_exponent_number.

   A number is either an integer or a decimal_point_number.

   The values of all types of numbers are to be interpreted in the normal way as base ten numbers.

6. Strings

   A string consists of a double quote <34>, followed by zero to 255 of the characters allowed by Section 3 in the I++ DME spec (excluding <34>, <13>, and <10>) followed by a double quote.

7. Tags

   There are two types of tags:

   CommandTag EventTag

   A CommandTag is formed by putting the digit zero <48> before an unsigned_four_digit_integer. This makes CommandTags look like five-digit integers, but that appearance is irrelevant. The command tag 00000 may not be used.

   An EventTag is formed by putting an upper case E <69> before an unsigned_four_digit_integer. The EventTag E0000 may not be used in a command string.

   Examples of CommandTags:

   04711 // tag is OK
   00020 // tag is OK
   20 // error; only 2 characters
   10710 // error; first character not zero
   00000 // error; explicitly disallowed.

   Examples of EventTags:

   E4711 // tag is OK
   E0333 // tag is OK
   e0333 // error; illegal first character, must use upper case E
   E0000 // error; explicitly disallowed
   E20 // error; only 3 characters
   A4711 // error; first character not E

   The first 5 characters of a command string represent a tag.

8. Command String Syntax

   A command string consists of the following, in order

   (a) a tag
   (b) a space
   (c) a method name
   (d) a left parenthesis <40>
   (e) an optional argument_list
   (f) a right parenthesis <41>
   (g) a carriage return line feed pair <13><10>

   An argument_list consists of one or more arguments separated by commas. A comma must not be placed after the last argument. One space may optionally follow each comma.

   An argument is a number, a string, or a method_identifier.

   A simple_method is a method name followed by a left parenthesis followed by either: a. a right parenthesis, or b. a number or a string followed by a right parenthesis.

   A method_identifier is one of the following:

   (a) a simple_method
   (b) a method name followed by a period followed by a simple_method

(c) a method name followed by a period followed by a method name followed by a period followed by a simple_method. Example: FoundTool.PtMeasPar.Speed()

(d) a method name followed by a period followed by a method name followed by a period followed by a method name followed by a period followed by a simple_method. Example: FoundTool.PtMeasPar.Speed.Min()

A method name is the name of one of the methods defined in Section 6.3 of the I++ DME Interface Specification version 1.1. Method names are not enclosed in quotes.

Only those combinations of method names and arguments defined in Section 6.3 of the I++ DME Interface Specification version 1.1 may be used together.

9. Examples

The following are example of potentially legal command strings. They may become illegal in context.

00001 StartSession()<13><10>

01095 GoTo(X(-0.75), Y(1.0), Z(-.00))<13><10>

07003 GetProp(FoundTool.PtMeasPar.Speed())<13><10>

7.4.2 Format of command string test files

1. What is an I++ DME Interface Command File?

An I++ DME Interface Command File is a file containing character strings to be stuffed into messages and sent to an I++ DME Interface server. Each character string represents a legal or illegal command message. The command file also contains character string separator sequences and end of file sequences that are not part of the character strings. In this spec "character string" will be used to refer to the characters that go into the message.

2. Suffix

Command files are identified by a ".prg" suffix.

3. Communications

It is assumed here that sockets are being used for communications. When sockets are used, the length of the character string being transmitted is given in the communication, and no terminator (such as a NULL) is used in the character string.

To be suitable for use with some other communications method, this file format may need to be modified.

4. Use of ASCII

All references to ASCII characters in this file spec are given using decimal (not octal or hex) numbers. An ASCII number enclosed in angle brackets (e.g., <13>) is used to represent ASCII characters.

5. How the Command File is Divided

The first character string of a command file starts with the first character in the file and ends on the last character before the first occurrence of two backslashes followed by a carriage return followed by a line feed (*i.e.*, <92><92><13><10>). The second character string starts with the next character in the file after that and ends on the first character before next occurrence of <92><92><13><10>, and so on. The <92><92><13><10> sequence is a separator and is not part of any character string.

The backslashes are used so that character strings representing illegal commands with <13><10> inside can be written in the file and used for testing.

To end the file, after the <92><92><13><10> following the last character string, there should be the sequence <58><13><10> <58><13><10>. This has the appearance of two lines each containing only a colon.

Using ASCII for non-printing characters, here is an example (written on two lines) of an entire command file with two character strings in it, each representing a legal command:

00001 StartSession()<13><10>\\<13><10>
00002 EndSession()<13><10>\\<13><10><58><13><10><58><13><10>

The first character string is: 00001 StartSession()<13><10> The second character string is: 00002 EndSession()<13><10>

When this file is viewed in most file viewers, it has the following appearance:

00001 StartSession()
\\
00002 EndSession()
\\
:
:

6. Legal Character Strings

   A legal character string consists of a legal command string, as defined in 7.4.1, and nothing else.

   A legal command string followed by any other characters before the separator forms an illegal character string.

7. Multiple Sessions

   Files of legal commands must always begin with a StartSession command. Files that represent complete sessions must always end with an EndSession command. Multiple sessions may be included in a single file.

8. Comments

   Command files contain no comments. This is to keep parsing easy. It is intended that a .txt (text) file with the same base name accompany each .prg file. The .txt file should explain the .prg file. It is suggested that the .txt file include every line of the .prg file.

   Reading of command files is expected to stop when the two colons ending the file are encountered. Thus, for most command file readers, anything after the colons is effectively a comment. Users who choose to do so can put comments there.

9. Examples

   In these examples, the character string separators are represented as \\ alone on a line, and it is assumed each line ends with <13><10>.

   (a) All Legal File

   In this file, all the character strings represent legal commands given in a legal order.

   StartSession()
   \\
   Home()
   \\
   GoTo(X(3), Y(-2.0))
   \\
   EndSession()
   \\
   :
   :

   (b) All Illegal Character Strings File

   In this file, all the characters strings are illegal.

   StartSession()
   hi mom
   \\
   home()
   \\
   GoTo(X3, Y-2)
   \\
   :
   :

   The first character string has "hi mom" after the first <13><10> and before the end of the string.
   The second character string spells the command "home" starting with a lower case h (must be upper case).
   The third character string fails to put the X and Y values in parentheses.

7.5 Using the Lego test artifact

7.5.1 How to purchase, assemble, and provide CAD support for the Lego artifact

The following three files are needed to purchase, assemble, and provide CAD support for the Lego artifact:

```
NISTI++DMEtestSuite1.1\testSuiteComponents\clientComponents\artifacts\lego\lego
shopping list.xls
NISTI++DMEtestSuite1.1\testSuiteComponents\clientComponents\artifacts\lego\
legoArtifactAssemblyInstructions.pdf
NISTI++DMEtestSuite1.1\testSuiteComponents\clientComponents\artifacts\lego\ProE
```

We now describe the contents of these two files and one folder

- The first file, lego shopping list.xls, consists of the following spreadsheets
  - A listing of the primary sets containing the required parts for the artifact
  - A listing of several alternate sets containing the required part(s) (in case the primary set containing the required part(s) becomes unavailable on www.lego.com)
- The second file, legoArtifactAssemblyInstructions.pdf, contains the assembly instructions for constructing the artifact from its component parts. These assembly instructions are automatically generated output from MLCAD, a CAD program for assembling Lego parts. MLCAD is built on top of LDraw, a low level part definition software[2]. The assembly instructions output of MLCAD give a clear and detailed sequence of artifact assembly, part by part. It also lists the part numbers and part names in the standard nomenclature of the LDraw specification.
- The directory, ProE, contains a representation of the artifact in a commercially available CAD format, which can be used to create further test files using appropriate simulation and program generation software.

The part names and numbers are listed at the end of every section of the assembly instructions (file name is legoArtifactAssemblyInstructions.pdf). Match these to the same names and numbers in the Lego shopping list spreadsheet in order to find in which set each particular part can be found. If you're lost, look at www.peeron.com, locate the set by the set number printed on the box (or bag), display the pictures of all the component parts of the set, and use those pictures to locate the correct part in the correct set.

7.5.2 How to set up the Lego artifact: definition of part origin and axis orientation

Referring to Figures 4 and 5, we define the part origin to be at the outside corner of the "tower" section (the section containing the highest point on the artifact), at the base of the entire artifact, with the x-axis along the long side of the tower, the y-axis along the short side, and the z-axis pointing up.

7.5.3 Reasons for the particular choice of parts on the Lego artifact

The Dimensional Metrology Equipment (DME) interface requires that the artifact allow the testing of simple motion and touch probing, as well as scanning, non-contact probing, probe clusters, wrist motion, and multiple carriage activity. The artifact has been designed with these activities in mind. Referring to Figures 4 and 5, we argue that

- The "Canopy Half Sphere 6 x 6 x 3 with Hinge" part on the artifact should be useful for testing scanning and synchronous wrist and carriage motion
- The presence of inner and outer cylinder parts and several inset locations should be useful for wrist motion and probe clusters
- The presence of several measurement locations that are widely spaced should be useful for testing multiple carriages
- There are several locations on the artifact suitable for use as a datum
- The "Cylinder 4 x 4 Hemisphere Multifaceted" part should be useful for exercising wrist motion and scanning given that the part has small planar facets with surface normal vectors at angles spanning roughly $\pm\pi$ radians in the both the x-z and y-z planes

[2] Both MLCAD and LDraw are freely available at http://www.lm-software.com/mlcad/ and http://www.ldraw.org/, respectively

- Cones and cylinders of various sizes and orientations, concentrically located on the artifact, may not be important for DME interface testing, but if a high level language like DMIS is used by the application software, these features could be helpful, too

The entire artifact must be secured with some kind of adhesive. Likewise, certain parts on the artifact will also need to be similarly secured, particularly the "Canopy Half Sphere 6 x 6 x 3 with Hinge" and the "Cylinder 4 x 4 Hemisphere Multifaceted."

We have written a file of I++ DME specification version 1.1 compliant command strings called, lego.prg (see Table 4), for execution with NIST's test suite utilities on the NIST Lego artifact.

7.5.4 Arguments for (and against) using the Lego artifact

In order to satisfy a variety of interface specification testing needs, an artifact has been designed at NIST consisting of a simple assembly of commercially available "components." The artifact is intended to be used for interface specifications not requiring high positional accuracy, but requiring high availability, malleability, and low cost. The artifact consists of an assembly of components or "parts." These parts can be found in "sets" that are available to be shipped within a day or two to any location worldwide from www.lego.com. The argument for using Lego parts and Lego sets is as follows:

- It will make the artifact easily and quickly obtainable
- The artifact can be easily and quickly modified
- Testing of interface specifications should not require measurement precision, so we can live with the kind of error that we would expect from artifact to artifact, however, we can still make each artifact quite stiff
- It will keep the price of the artifact down
- We can approximate several of the features we would expect to encounter in automotive/aerospace parts

Even though the accuracy in each instance of the artifact will vary substantially more than other commonly employed test artifacts, effort has been made to make the artifact as stiff as possible, partly through the use of two layers of large "brick" parts on the bottom to provide stiffness in the foundation.

There are some negatives to using Legos, namely,

- Cannot realize certain features such as fillets
- Artifact to artifact errors will be relatively large, due to the manual construction
- Sets currently available worldwide on www.lego.com may become unavailable at a later date

We have tried to mitigate some of these negatives as follows:

- We have found and list all currently available sets containing each required part to allow alternatives if the primary set becomes unavailable
- We have chosen parts from sets that are reasonably expected to be perennially available on www.lego.com, e.g., bulk parts seem to be consistently available worldwide
- For parts in those sets that may suddenly become unavailable, we plan to keep updating the artifact and publishing any changes to all users
- Depending on the usefulness to the community of this artifact, we may choose to "stockpile" those parts we anticipate may suddenly become unavailable

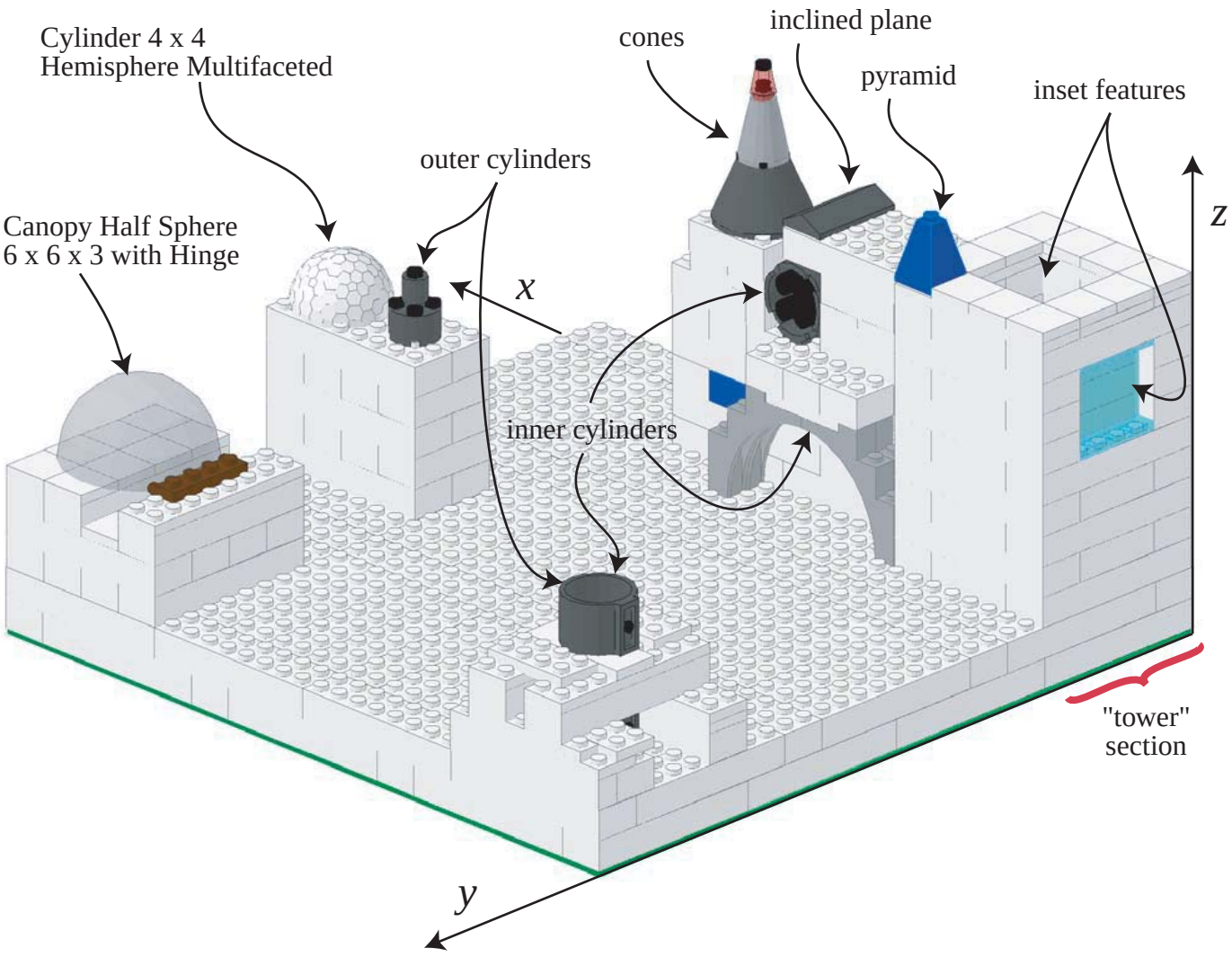

Figure 4: The Lego artifact: viewpoint #1

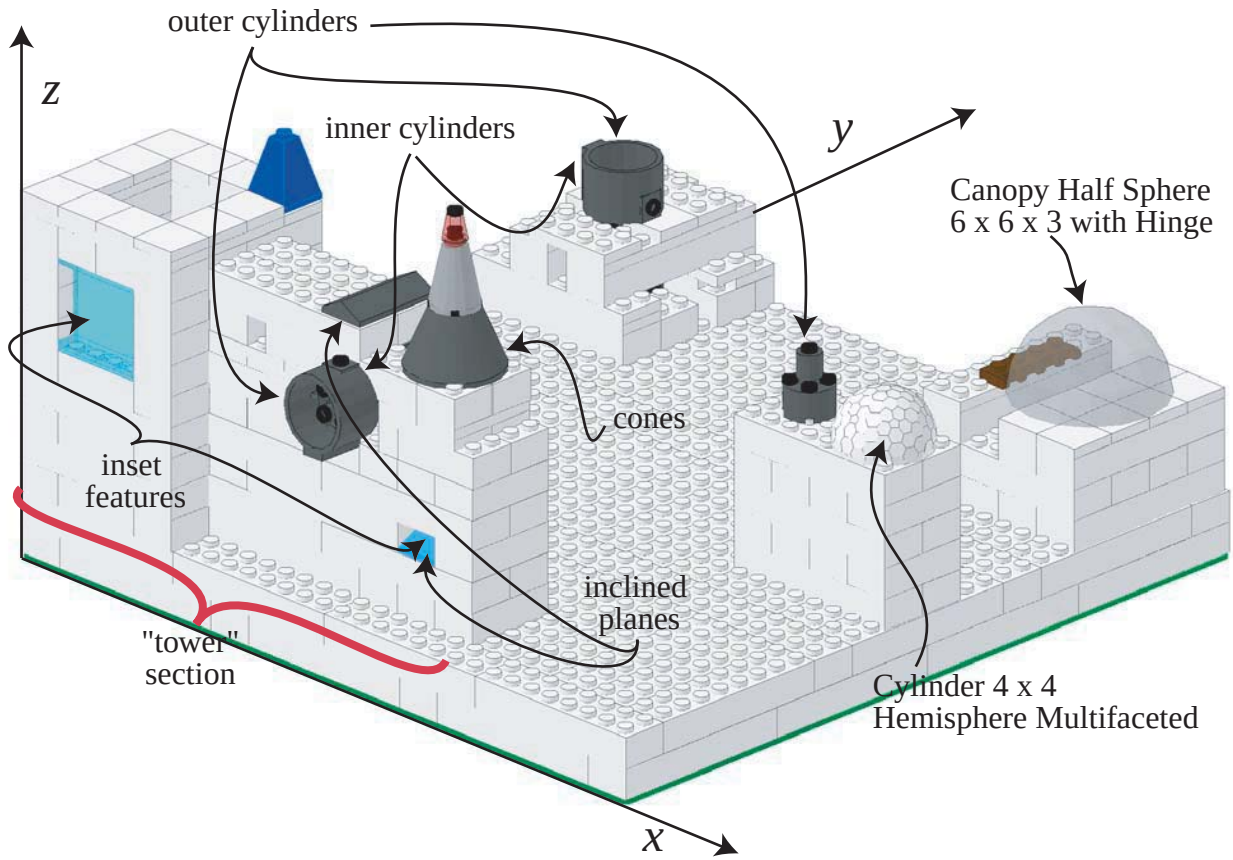

Figure 5: The Lego artifact: viewpoint #2

7.6 Using the response parser code

The client-side utility executable contains a response parser. However, another response parser, one that has not yet been integrated into the client-side executable, has been developed and tested, one that is more consistent with the command parser. Developers are encouraged to utilize this latter resource as a stand-alone parser for I++ response files to check that responses are parsable. We plan in our next version to integrate this parser into the client-side utility executable.

All files relating to the PC and UNIX versions, respectively, of the response parser can be found in

```
NISTI++DMEtestSuite1.1\standAloneTestSuiteComponents\ParserResPC\
NISTI++DMEtestSuite1.1\standAloneTestSuiteComponents\ParserResUNIX\
```

and all directories listed in this section are subdirectories of ParserResPC only, since the UNIX version has only minor modifications. The PC software compiles under Visual C++ (version 6.0 Professional) consistent with our platform constraints (see Section 3).

7.6.1 Using the response parser as stand-alone support for server-side implementations

An executable for the stand-alone response parser, which should run on any Windows operating system, is bin\parserRes.exe. Sample response files are in \test_files. Use the executable with a command as follows:

```
bin\parserRes.exe <file name>
```

with your file name substituted for <file name>.

for example,

```
bin\parserRes.exe test_files\all_res_ok.res
```

Descriptions of the format of response strings and response string files are in Sections 7.6.3 and 7.6.4.

The test_files subdirectory contains only two test programs, all_ok.prg and checker_errors.prg. More test files may be available in other directories in this distribution, but check to be sure the format is as described in Section 7.6.4.

The main function reads response strings from the response file, and calls the parser's parseResponse method. If parsing succeeds, parseResponse makes an instance of a Response and returns a pointer to it. The main function then prints the response followed by \\ on a separate line. If parsing fails, parseResponse returns a NULL pointer. The main function then prints the text of the response string followed by the error message caused by the response string. To test your own responses, put them in a file in the required format, and give the shell command described above.

7.6.2 Integrating the response parser in a client-side implementation

To use the response parser in your own client-side implementation, the setInput() method must first be called to copy an input string into the parser's inputArray.

The caller should then call parseResponse(). If there is no error in the input string, parseResponse makes an instance of a Response from the input string and returns a pointer to that instance. If there is an error, parseResponse returns a NULL pointer.

After calling parseResponse, the caller should call getParserErr() to see if there was an error. If the returned error code is not OK, the caller can call getErrorMessage(code) to get an error message that describes the error. If the error code is OK, the caller can start processing the Response instance. The caller may call parseTag() before calling parseResponse(), but that is not required.

Now we look at the syntax of individual response strings.

7.6.3 Format of response strings

General issues relating to response strings: Any string purported to be a response string but failing to conform to the requirements given here should cause an error in the client.

1. What is an I++ response string?

   An I++ response string is a string of characters intended to be put into a character string to be transmitted by an I++ DME Interface server through a communications system to a client. The response string represents an I++ DME Interface response.

2. Use of ASCII (American Standard Code for Information Interchange)

   The use of ASCII is as described in section 7.4.1. In this file, an ASCII character may be denoted by a decimal integer in angle brackets. For example, <32> is the ASCII space character.

3. Character set

   The use of characters is as described in section 7.4.1.

4. Response string length

   The number of characters in a response string, including the <13><10> at the end, must not exceed 256. This differs from the length requirements for commands.

5. Numbers

   The format of numbers is as described in section 7.4.1.

6. Strings

   The format of strings is as described in section 7.4.1.

7. Tags

   The format of tags is as described in section 7.4.1. The EventTag E0000 may be used in a response string to indicate errors that are not the result of executing a command or cannot be identified with a specific command.

   If a response can be identified with a command, the tag in the response must be the same as the tag in the command.

8. Commas

   Whenever a comma <44> is used, it may optionally be followed by a single space <32>. Spaces may not be used anywhere else. Wherever the use of a comma is described below, it is implicit that the optional space may be used.

9. Response string syntax

   A response string consists of the following, in order

   (a) a tag.
   (b) a space.
   (c) a single character that is a response type indicator. This must be one of &<38>, %<37>, #<35>, or !<33>.
   (d) zero to many continuation characters as described below.
   (e) a carriage return line feed pair <13><10>.

   If the response type indicator is & or %, there are zero continuation characters.

   If the response type indicator is ! or #, the first continuation character is always a space.

Error response: If the response type indicator is !, this indicates an error. The additional continuation characters consist of the following, in order.

1. "Error" (without the quotes).
2. a left parenthesis <40>.
3. a single character that must be one of 1 2 3 9. This character must be the severity character as given in section 8.2 of the spec for the error number described in e (below).
4. a comma.
5. a number consisting of four digits (with no sign and no decimal point) that is one of the error numbers given in section 8.2 of the spec.
6. a comma.
7. a string.
8. a comma.
9. a string. This string must be the text as given in section 8.2 of the spec for the error number described in item 5 (above).
10. a right parenthesis <41>

Data response: If the response type indicator is #, this indicates data is being returned. The additional continuation characters are one of following.

1. AlignTool Data

   This is used only in response to an AlignTool command. It has the following, in order.

   (a) A left parenthesis.
   (b) either two or five numbers, each followed by a comma.
   (c) a number.
   (d) a right parenthesis.

   Example 1a: 00001 # (1.000000,0.000000,0.000000)
   Example 1b: 00001 # (1.00000, 0.00000, 0.00000, 0.00000, 0.00000, 1.00000)

2. Property type data

   This is used in response to either an EnumProp or and EnumAllProp command. It has the following, in order:

   (a) a string giving the name of a property. This must be one of: "Tool", "FoundTool", "GoToPar", "PtMeasPar", "Speed", "MaxSpeed", "MinSpeed", "Accel", "MaxAccel", "MinAccel", "Approach", "Retract", or "Search".
   (b) a comma.
   (c) a string giving the name of a data type. This must be one of: "Number", "Property", "String". If item a above is "Tool", "FoundTool", "GoToPar", or "PtMeasPar", this must be "Property". If item a above is "Speed", "MaxSpeed", "MinSpeed", "Accel", "MaxAccel", "MinAccel", "Approach", "Retract", or "Search", this must be "Number".

   Example 2a: 00001 # "Speed", "Number"
   Example 2b: 00001 # "Tool","Property"

3. Coordinate System Type Data

   This is used only in response to a GetCoordSystem command. It has the following, in order:

   (a) "CoordSystem" (without the quotes).
   (b) a left parenthesis.
   (c) one of: MachineCsy, MoveableMachineCsy, MultipleArmCsy, PartCsy.
   (d) a right parenthesis.

   Example 3a: CoordSystem(MachineCsy)

4. Coordinate System Transformation Data

   This is used only in response to a GetCsyTransformation command. It has the following, in order:

   (a) "GetCsyTransformation" (without the quotes).
   (b) a left parenthesis.
   (c) one of: PartCsy, JogDisplayCsy, JogMoveCsy, SensorCsy, MultipleArmCsy.
   (d) six numbers, each preceded by a comma.
   (e) a right parenthesis.

   Example 4a: 00001 # GetCsyTransformation(PartCsy,1, -2.00,3, 30.0, 45.0,20.0)

5. Error Information Data

   This is used only in response to a GetErrorInfo command. It consists of a single string. A single GetErrorInfo command may elicit several responses.

   Example 5a: 00001 # "no clue"

6. Error Status Data

   This is used only in response to a GetErrStatusE command. It consists of the following, in order:

   (a) "ErrStatus" (without the quotes).
   (b) a left parenthesis.
   (c) a single character that is either 0 or 1.
   (d) a right parenthesis.

   Example 6a: E0001 # ErrStatus(0)

7. Machine Class Data

   This is used only in response to a GetMachineClass command. It consists of the following, in order:

   (a) "GetMachineClass" (without the quotes).
   (b) a left parenthesis.
   (c) the string "CartCMM" (with the quotes).
   (d) a right parenthesis.

   Example 7a: 00001 # GetMachineClass("CartCMM")

8. Is Homed Data

   This is used response to an IsHomed command or a GetXtdErrStatus command. It consists of the following, in order:

   (a) "IsHomed" (without the quotes).
   (b) a left parenthesis.
   (c) a single character that is either 0 or 1.
   (d) a right parenthesis.

   Example 8a: 00001 # IsHomed(1)

9. Is User Enabled Data

   This is used in response to an IsUserEnabled or a GetXtdErrStatus command. It consists of the following, in order:

   (a) "IsUserEnabled" (without the quotes).
   (b) a left parenthesis.
   (c) a single character that is either 0 or 1.
   (d) a right parenthesis.

   Example 9a: 00001 # IsUserEnabled(1)

10. Property data

    This is used in response to a GetProp, GetPropE, or SetProp command. It consists of the following in order:

    (a) the keyword "Tool" or "FoundTool" (without the quotes).
    (b) a dot <46>.
    (c) the keyword "PtMeasPar" or "GoToPar" (without the quotes).
    (d) a dot.
    (e) one of the following keywords (without the quotes): "MaxSpeed", "Speed", "MinSpeed", "MaxAccel", "Accel", or "MinAccel". If the preceding keyword is "PtMeasPar", this may also be "Approach", "Retract", or "Search" (without the quotes).
    (f) a left parenthesis.

(g) a number.

(h) a right parenthesis.

Example 10a: 00001 # Tool.PtMeasPar.MinAccel(2.000000)

11. Position data

This is used in response to a Get, PtMeas, or OnMoveReportE command. It consists of one to three parts separated by commas. Each part consists of the following, in order:

(a) the character X, Y, or Z.

(b) a left parenthesis.

(c) a number.

(d) a right parenthesis.

Each of X, Y, and Z may appear at most once, but they may appear in any order.

Example 11a: 00001 # Y(2.000000), Z(3.000000)
Example 11b: 00001 # X(1.2)
Example 11c: 00001 # X(1.2),Z(3.0), Y(2.0)

7.6.4 Format of response string test files

1. What is an I++ DME interface response file?

An I++ DME Interface Response File is a file containing character strings to be stuffed into messages and sent to an I++ DME Interface client. Each character string represents a legal or illegal response message. The response file also contains character string separator sequences and end of file sequences that are not part of the character strings. In this spec "character string" will be used to refer to the characters that go into the message.

2. Suffix

Response files are identified by a ".res" suffix.

3. Communications

It is assumed here that sockets are being used for communications. When sockets are used, the length of the character string being transmitted is given in the communication, and no terminator (such as a NULL) is used in the character string.

To be suitable for use with some other communications method, this file format may need to be modified.

4. Use of ASCII

All references to ASCII characters in this file spec are given using decimal (not octal or hex) numbers. An ASCII number enclosed in angle brackets (e.g., <13>) is used to represent ASCII characters.

5. How the Response File is Divided

The first character string of a response file starts with the first character in the file and ends on the last character before the first occurrence of two backslashes followed by a carriage return followed by a line feed (*i.e.*, <92><92><13><10>). The second character string starts with the next character in the file after that and ends on the first character before next occurrence of <92><92><13><10>, and so on. The <92><92><13><10> sequence is a separator and is not part of any character string.

The backslashes are used so that character strings representing illegal responses with <13><10> inside can be written in the file and used for testing.

To end the file, after the <92><92><13><10> following the last character string, there should be the sequence <58><13><10> <58><13><10>. This has the appearance of two lines each containing only a colon.

Using ASCII for non-printing characters, here is an example (written on two lines) of an entire response file with two character strings in it, each representing a legal response:

00001 &<13><10>\\<13><10>
00001 %<13><10>\\<13><10><58><13><10><58><13><10>

The first character string is: 00001 &<13><10>
The second character string is: 00001 %<13><10>

When this file is viewed in most file viewers, it has the following appearance:

00001 &
\\
00001 %
\\
:
:

6. Legal Character Strings

   A legal character string consists of a legal response string, as defined in 7.6.3, and nothing else.

   A legal response string followed by any other characters before the separator forms an illegal character string.

7. Comments

   Response files contain no comments. This is to keep parsing easy. It is intended that a .txt (text) file with the same base name accompany each .res file. The .txt file should explain the .res file and the corresponding .prg file, if there is one.

   Reading of response files is expected to stop when the two colons ending the file are encountered. Thus, for most response file readers, anything after the colons is effectively a comment. Users who choose to do so can put comments there.

8. Examples

   In these examples, the character string separators are represented as \\ alone on a line, and it is assumed each line ends with <13><10>.

   (a) All Legal File

   In this file, all the character strings represent legal responses.

   00001 &
   \\
   00012 # IsHomed(1)
   \\
   00015 # Y(2.000000), Z(3.000000)
   \\
   00015 %
   \\
   :
   :

   (b) All Illegal Character Strings File

   In this file, all the characters strings are illegal.

   00001 &
   oops
   \\
   00003 # (1.0, 2.0)
   \\
   00004 # Isuserenabled(1)
   \\
   :
   :

The first character string has "oops" after the first <13><10> and before the end of the string.

The second character string does not correspond to any valid response format.

The third character string has two lower case letters where they should be upper case in IsUserEnabled.

8 Using server-side components for facilitating client-side implementations

The server-side components of the test suite are listed and briefly described in Table 2. We will now describe how to use these components in your implementations of the I++ DME specification.

8.1 Operating the server-side GUI front end

The file, serverDlg.cpp, contains code for the server-side GUI front end and it also contains much of the "glue" code for the server-side utility. The location of serverDlg.cpp is

```
NISTI++DMEtestSuite1.1\testSuiteComponents\serverComponents\src\Server
```

Since the server-side GUI is set up to receive commands from the client-side, the user needs to set up and execute the server-side GUI first, (to execute, just double click the server.exe icon). Once executing, this GUI will wait indefinitely for a connection from the client. Server-side GUI setup involves the following steps (refer to Figure 6 for a picture of the server-side GUI):

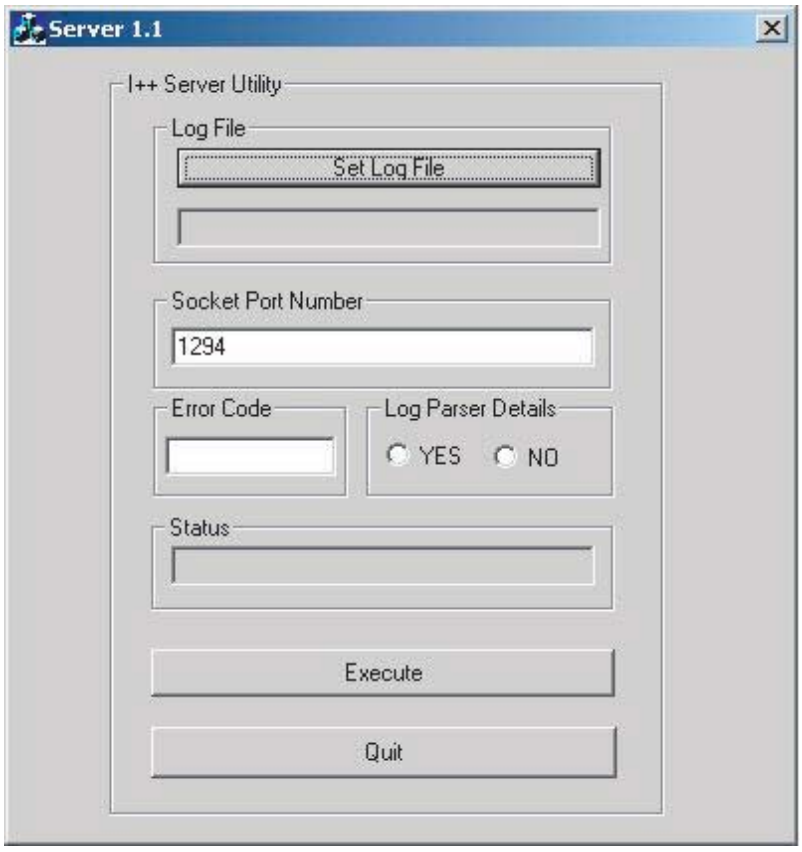

Figure 6: The server-side GUI front end

- If you wish to log test result information, click the "Set Log File" button and provide a file name. It is suggested that you set up a folder to store the log files for different test files. Command and error information are recorded in these log files. Selecting this option also specifies error logging for parsing and context checking errors. These errors are more detailed than the I++ specification compliant errors that are available as responses. They reveal more detail about why the command string parsing failed or why the command had an inappropriate context. For more detail on server-side log files, see Section 8.2.
- Select the port number for TCP/IP socket communications. This number must match the one chosen on the client-side. The default port number is 1294, which is the one internationally defined for this type of connection. Therefore, it should be used whenever possible. However, other port numbers may work as long as the client-side socket uses the same numbers.
- Select the number corresponding to the desired server-side test case, which currently defaults to 0, where 0 means "all correct responses." Additional server-side test cases are planned that will generate a variety of illegal responses in order to more fully test client-side implementations.

- To execute, click the Execute button. At such point, the Status display would show "Waiting for Connection" until the client site connects, and, at such point, the server site Status display would show "Client Connected." This information continues throughout the execution of the entire test file.
- Click the Quit button once the command file has been completely executed. Only this action would cause the logged information to be dumped to the log file. The program completion message is displayed in Status display on the client side GUI .

8.2 Server-side utility

The server-side utility is the component consisting of mostly "glue" code that integrates all the other server-side components into a single executable. The server-side utility executes the socket read of commands, socket write of responses, command string parsing, command context checking, and command execution (*i.e.*, coarse CMM simulation). The operation of these components is illustrated in Figure 2. It also maintains "world model" data and methods files (world.h and world.cpp) that contain data about the virtual CMM and about the state of the system (*e.g.*, the context of commands) and methods needed to update and maintain that data in world.cpp.

There are two execution threads in the server-side utility. Each thread represents a cyclically executing piece of code. The contents of the first thread are as follows:

1. Check the socket for data
2. If there is data in the socket, the data is parsed and checked for a correctly formed tag and also to ensure that the tag number is not currently in use.
3. If the tag is legal, the command string is put in either the fast or slow queue.
4. If the tag is illegal, an error response is generated (and logged, if that option is selected) and a response string is put in the response queue.
5. The response queue is checked to see if it has responses to send back to the client; if it does, then the response queue is emptied, sending the response strings to the client.
6. The code waits until the end of the sampling period for this thread, then returns to step number 1.

The contents of the second thread are as follows:

1. Check the fast queue for a command string.
2. If there is no command string in the fast queue, it checks the slow queue for a command string.
3. If a command was found from either queue and a multicycle command (that is, AlignTool(), Home(), GoTo(), PtMeas(), or PtMeasIJK()[3]) is not currently executing, the parser is called to check the command for validity (*i.e.*, the parser checks for syntax errors and parameter errors).
4. If a multicycle command is currently executing and there is a command on the fast queue, the parser is called to check the command for validity.
5. If a multicycle command is currently executing and there is a command on the slow queue, but none on the fast queue, the queues are left alone and the command is not parsed.
6. If the command string is not valid, an error response is generated and a response string is put in the response queue.
7. If the command string is valid, the parser returns a command object (*i.e.*, an instance of the appropriate command class).
8. If AbortE() is received and valid, both command queues are cleared.
9. The command object is sent to the context checker.
10. If the command has bad context, an error response is generated and a response string is put in the response queue.
11. If the context of the command is good, the command object is given to the executor.

[3] actually PtMeasIJK() is not available in the server-side utility, since it appears to be identical to PtMeas() in the I++ DME spec version 1.1

12. The executor is checked for responses, and any responses are sent to the response queue.

13. The code waits until the end of the sampling period for this thread, then returns to step number 1.

Having two threads of execution allows us to set a unique cycle time (*i.e.*, sampling rate) for the execution of each thread. Typically we want to have the first thread at a higher sampling rate and the sampling rate of the second thread be no less than the rate of the first, since there is no need to check the command queues, if you have not read the socket for a new command since you last checked the command queues for the presence of commands. Currently, the first thread has a sampling period of 20 ms and the second thread, 50 ms.

If the logging option is selected in the GUI (see Section 8.1), the log file will include information on commands received and errors detected. The log file has the following general format:

- Timestamp, received command tag number, command string.
- Timestamp, serial number of the command, acknowledgement symbol, &.
- Timestamp, tag number of the command, followed by either only a command completion symbol, or any other additional information that the I++ DME specification calls for after the completion of a command.
- Timestamp, tag number of the command, error report for the command.
- If there is an error in the command string (detected by the command parser), both the I++ error type and a more detailed description of the error are logged.
- If there is an error in the command context, a detailed description of that error is logged.

The log file may include additional execution status information users have programmed into the server utility. For example, current tool position, execution state, etc. This will be explained further in Section 8.7 on the executor.

8.3 TCP/IP socket read and write

This code is virtually identical to the socket read/write component in the client-side components as described in Section 7.2.

8.4 Using the command string parser

The command string parser is a separately defined software component with precisely specified interfaces. A stripped-down version of the parser has been integrated with the overall server-side utility (Section 8.2) and it also exists in a stand-alone version (with a main()). Developers are encouraged to utilize the stripped-down and stand-alone versions of the command parser in the following ways,

- In the stripped-down version as part of the server-side utility for facilitating client-side implementations
- In the stripped-down version for integration into an implementor's server-side implementation
- As a stand-alone version for I++ command files to test if input commands are parsable.

Descriptions of the format of command strings is in Section 7.4.1 and command string files is in Section 7.4 under the heading, Command test file format.

All files relating to the stripped-down and stand-alone versions of the parser, respectively, described in this section can be found in

```
NISTI++DMEtestSuite1.1\standAloneTestSuiteComponents\ParserCmdPC
NISTI++DMEtestSuite1.1\testSuiteComponents\serverComponents\ParserCmdUNIX
```

and all directories listed in this section are subdirectories of it. We will describe the command parser for PC only (though one exists for UNIX targets as well. The parserCmd.cc file will compile under GNU g++. The parserCmd.cpp file will compile in Visual C++. The only difference between parserCmd.cc and parserCmd.cpp is that #ifdef PARSER_MAIN and the matching #endif have been commented out in parserCmd.cpp.

The file, parser.cpp, includes documentation giving the rules for parsing the command strings for each I++ command. The documentation of the parserCmd class in source/parserCmd.h describes how the parser is intended to be used. The documentation of main in source/parserCmd.cc describes how main uses it.

A UNIX version of the stand-alone parser exists. The UNIX executable (for SUN systems) is bin/parserCmd. Sample command files are in ../I++test_files. Use the executable with a command like: bin/parserCmd ../I++test_files/all_ok.prg.

8.5 CMM and tools related components

In order to further modularize the server-side components, we organized the various variables (and the methods required to maintain those variables) relating to the CMM system and its environment (world) into a separate set of source code files, world.h, world.cpp, and tools.h. These files are contained in the directory

`NISTI++DMEtestSuite1.1\testSuiteComponents\serverComponents\src\CMM`

The file, world.h, defines data members and methods for a world modeler that keeps track of the state of a system executing I++ DME commands. The methods in executor.cpp and serverDlg.cpp use many of the methods and data definitions in world.h and world.cpp. serverDlg.cpp is the file for the server-side GUI front end and contains much of the "glue" code for the server-side utility.

For those I++ DME specification server-side implementors who wish to integrate world.h into their own server code, we encourage them to use world.h and world.cpp as a template, *i.e.*, to use some of the constructs, replace some of the constructs, and add new constructs.

These files contain a substantial amount of code to support coordinate system transformations. However, we expect that many developers implementing I++ DME on the server-side will want to keep their proprietary version of world.h, world.cpp, and tools.h. Nonetheless, world.h, world.cpp, and tools.h are required for operation of the server-side utility.

8.6 Command context checker

The command context checker is a separately defined software component with precisely specified interfaces. The context checker maintains a record in the world model (world.cpp and world.h) of the previous commands. Using constraints in the specification, the context checker determines the legality of the current command, given its context.

A stripped-down version of the context checker has been integrated with the overall server-side utility (Section 8.2) and it also exists in a stand-alone version, *i.e.*, with a main(). The main() function uses both the command parser and the checker so that it can be used with the test files of Section 7.4.2. Developers are encouraged to utilize the stripped-down and stand-alone versions of the command context checker in the following ways,

- In the stripped-down version, as part of the server-side utility, for facilitating testing client-side implementations
- In the stripped-down version for integration into an implementor's server-side implementation
- In the stand-alone version for I++ command files to check that commands are parsable.

Descriptions of the format of command strings is in Section 7.4.1 and command string files is in Section 7.4.2 under the heading, Command test file format.

All files relating to the stripped-down and stand-alone versions of the context checker for the PC, respectively, described in this section can be found in

`NISTI++DMEtestSuite1.1\standAloneTestSuiteComponents\CheckerCmdPC`
`NISTI++DMEtestSuite1.1\testSuiteComponents\serverComponents\CheckerCmdUnix`

A command context checker exists in both PC and UNIX (SUN systems) formats. The UNIX version of the context checker, checker.cc, will compile under GNU g++ and is located among the stand-alone components. The PC version of the context checker, checker.cpp, will compile in Visual C++ and is located among both the stripped-down and stand-alone components. The only difference between checker.cc and checker.cpp is that #ifdef CHECKER_MAIN and the matching #endif have been commented out in checker.cpp.

The file, checker.cpp, defines functions in the checker class and defines a main function outside the checker class. It also includes documentation giving the rules for parsing the command strings for each I++ command. Functions named checkXXX (where XXX is a command name) are used to check each type of command in context. The arguments to each command are assumed to have passed the checks performed by the parser. The documentation for each of these functions gives the rules that it is enforcing and gives one or more references to pages of version 1.1 of the I++ DME specification. If this checker is used without the checks performed by the parser having been made previously, the checker may crash or give wrong results.

Several semantic checks are made in the context checker that could be made in the parser (because context is not required). For example, the check that a direction vector is not (0, 0, 0). These are identified in checker.cpp as semantic checks.

The reference pages in checker.cpp reference both text and examples. Text references are given in parentheses. Example references are given in brackets. Other references are not enclosed. For example: Reference pages: 21 23 (35) [36] 63 means there

is relevant text on page 35 and an example on page 36. Reference page 14 is not referenced because the print is too small and page 22 is not referenced since everything on it appears identically elsewhere.

The checker class, defined in checker.h, should make it self-evident how the checker component is intended to be used. The documentation of main in source/checker.cc describes how main uses the checker class.

The executables for PC and UNIX, respectively, are

```
NISTI++DMEtestSuite1.1\standAloneTestSuiteComponents\CheckerCmdPC\bin\checkerCmd.exe
NISTI++DMEtestSuite1.1\standAloneTestSuiteComponents\CheckerCmdUnix\bin\checkerCmd
```

Sample command files for use with the context checker are in ../I++test_files. Use the executable with a command like: bin/checker ../I++test_files/all_ok.prg.

8.7 Command executor (CMM simulator)

The command executor is a separate software component with separate source code and clearly defined interfaces. The command executor is integrated into the server-side utility. The command executor acts in the following roles:

- As a coarse CMM simulator
- As a locus for generating server-side test cases (generally erroneous responses of various types)
- As a separate component, so that the executor, of all the components linking into the server-side utility, is the only one that would need to be replaced by proprietary code in order to develop a server-side implementation. This should be clear from Figure 2.

8.8 Command and response C++ classes

A common set of command and response classes does not seem to be required by the I++ DME specification, however their use by all implementors is highly encouraged, in order to achieve a high level of system interoperability in the end. Using common command and status classes in implementations will reduce development and debug time and will streamline the testing and analysis process.

NIST has defined a set of I++ DME specification compliant command and response classes. Accompanying these classes are C++ files defining various methods for each of these classes. In order to simplify the class structure, the actual command and response classes are derived classes from command and response base classes, respectively. The primary function of these classes is to provide a common set of data structures for passing data and generating command and response strings. The role of the command and response classes is illustrated in Figure 2. Also defined are classes for handling data types and errors defined within the specification. The data classes contain the necessary logic for formatting the data according to the specification when a command or response string is being generated. Unified Modelling Language (UML) formatted diagrams can be seen in Figures 7 and 8.

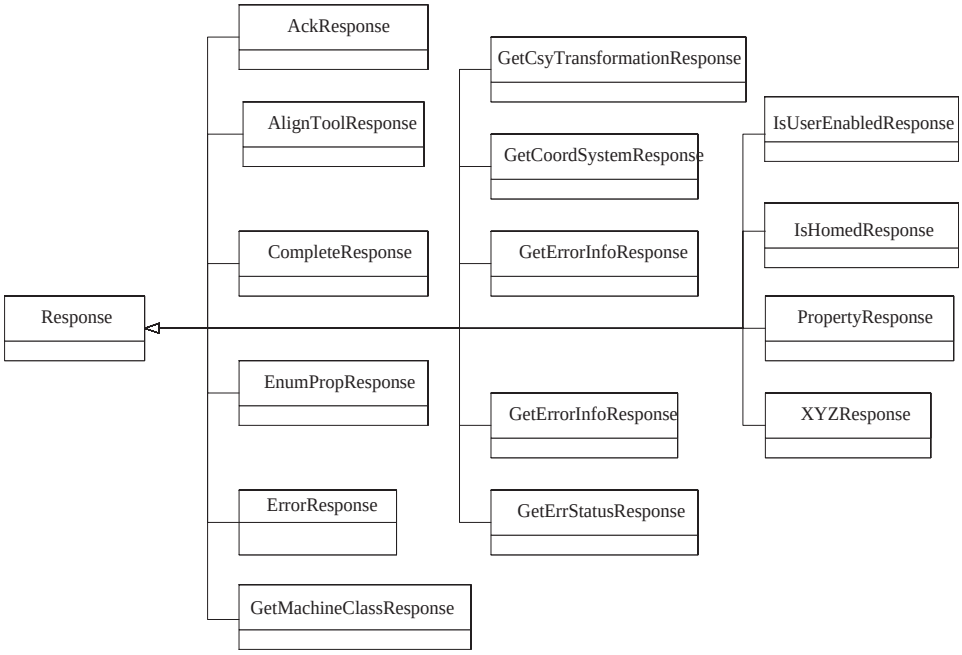

Figure 7: UML diagram for I++ response classes

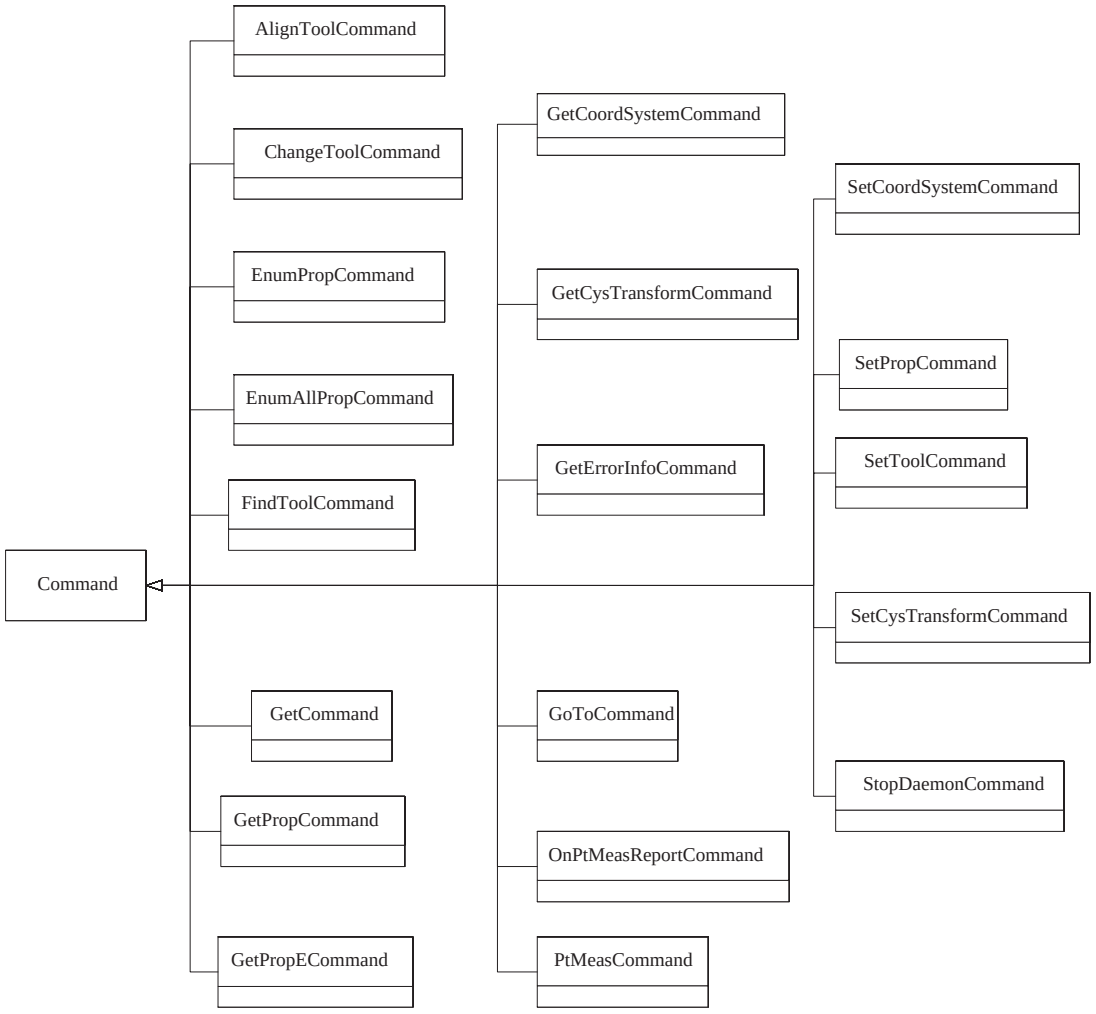

Figure 8: UML diagram for I++ command classes